Пермский филиал федерального государственного автономного
образовательного учреждения высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет экономики, менеджмента и бизнес-информатики

Виноградов Никита Андреевич

ОРГАНИЗАЦИЯ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ. ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ ИНТЕРПРЕАТОР И НАБЛЮДАТЕЛЬ

Лабораторная работа

студента образовательной программы «Программная инженерия»
по направлению подготовки *09.03.04 Программная инженерия*

Руководитель
к.т.н., доцент кафедры Информационных технологий в бизнесе НИУ ВШЭ-Пермь

А.В. Кычкин

Пермь, 2020 год

Оглавление

Глава 1. Интерпретатор

Поведенческий шаблон проектирования, решающий часто встречающуюся, но подверженную изменениям, задачу.

1.1. Назначение

- Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.
- Отображает проблемную область в язык, язык – в грамматику, а грамматику – в иерархии объектно-ориентированного проектировани.

1.2. Структура

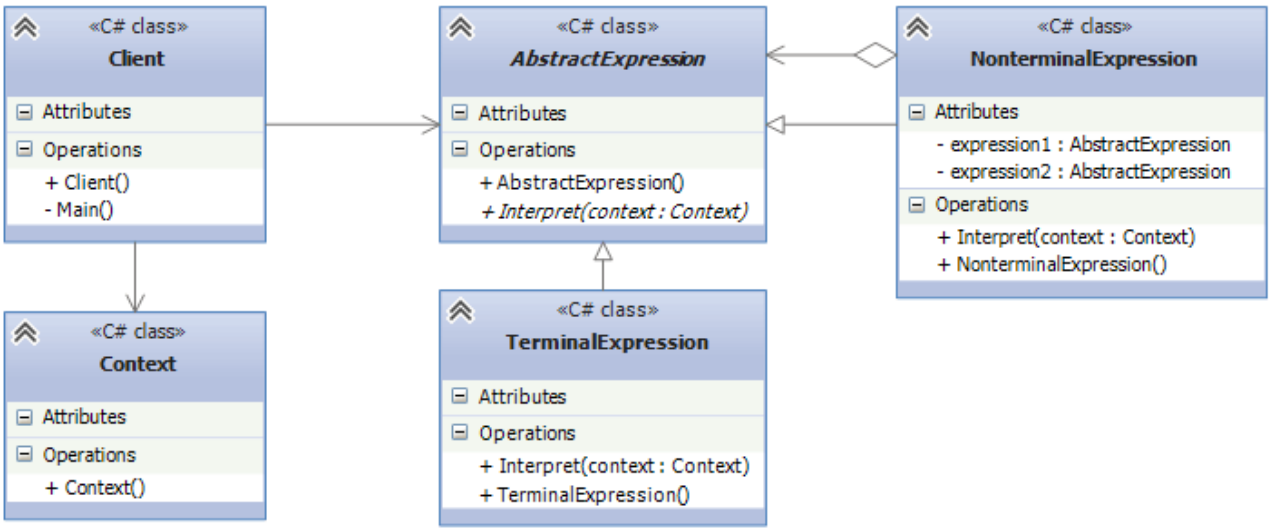

Рис. 1.1. Структура классов паттерна Заместитель

Участники:

- *AbstractExpression* - определяет интерфейс выражения, объявляет метод Interpret().
- *TerminalExpression* - терминальное выражение, реализует метод Interpret() для терминальных символов грамматики. Для каждого символа грамматики создается свой объект TerminalExpression.

- *NonterminalExpression* - нетерминальное выражение, представляет правило грамматики. Для каждого отдельного правила грамматики создается свой объект NonterminalExpression.

- *Context* - содержит общую для интерпретатора информацию. Может использоваться объектами терминальных и нетерминальных выражений для сохранения состояния операций и последующего доступа к сохраненному состоянию.

- *Client* - строит предложения языка с данной грамматикой в виде абстрактного синтаксического дерева, узлами которого являются объекты TerminalExpression и NonterminalExpression.

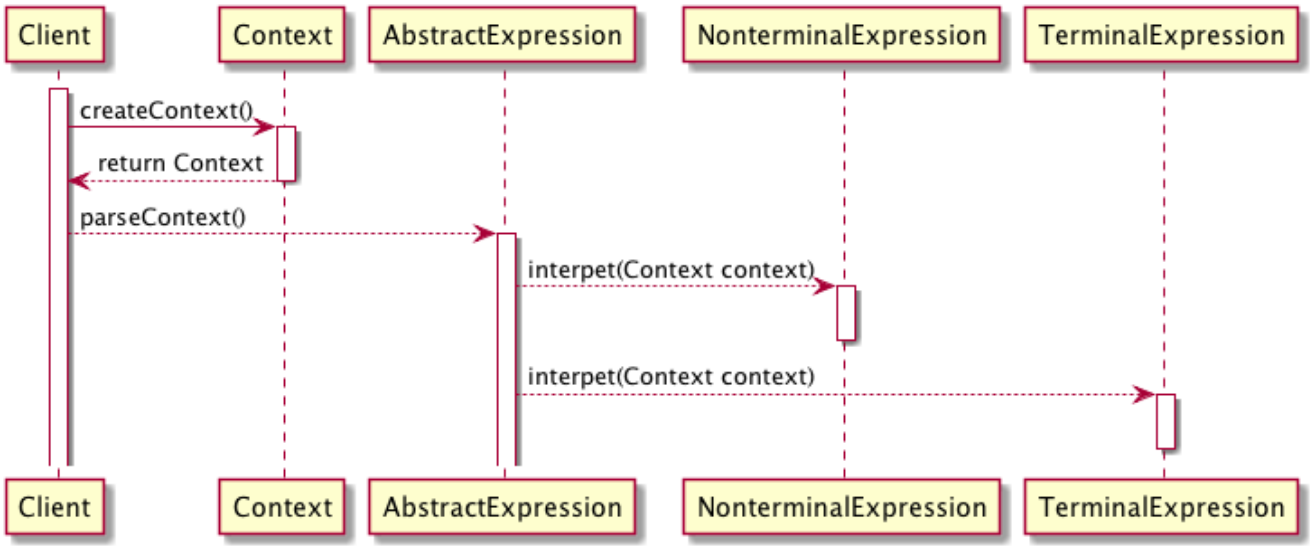

Рис. 1.2. Диаграмма последовательности паттерна Заместитель

1.3. Способ применения

Пусть в некоторой, хорошо определенной области периодически случается некоторая проблема. Если эта область может быть описана некоторым “языком“, то проблема может быть легко решена с помощью “интерпретирующей машины“.

Глава 2. Реализация паттернов

2.1. Диаграмма классов

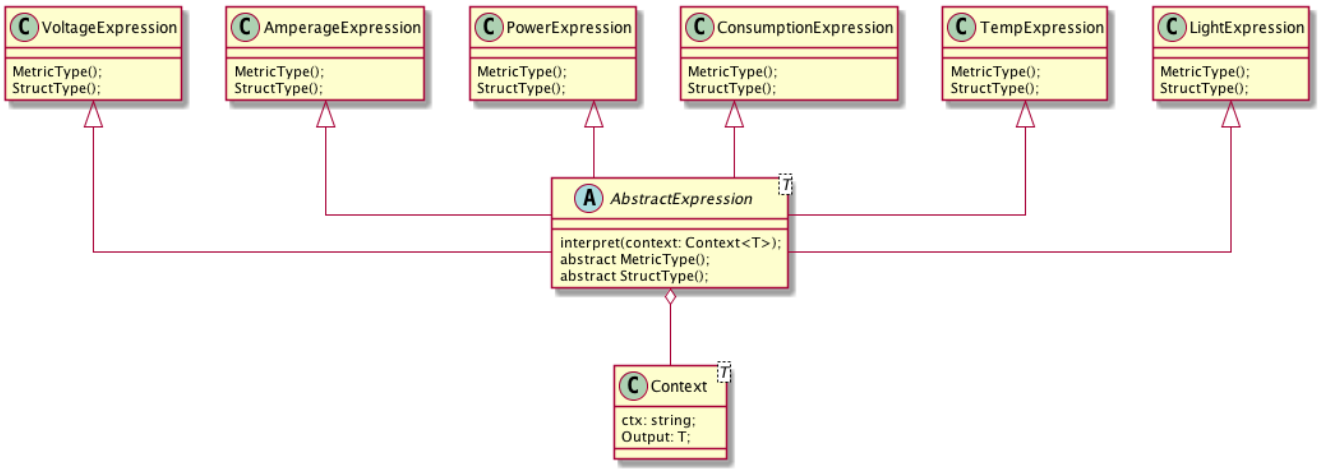

Рис. 2.1. Диаграмма классов паттерна Интерпретатор

Участники:

- *AbstractExpression* - определяет интерфейс выражения, объявляет метод Interpret().
- *Context* - терминальное выражение, реализует метод Interpret() для терминальных символов грамматики. Для каждого символа грамматики создается свой объект.
- *AmperageExpression* - терминальное выражение, реализует метод Interpret() для терминальных символов грамматики. Для каждого символа грамматики создается свой объект.
- *PowerExpression* - терминальное выражение, реализует метод Interpret() для терминальных символов грамматики. Для каждого символа грамматики создается свой объект.
- *ConsumptionExpression* - терминальное выражение, реализует метод Interpret() для терминальных символов грамматики. Для каждого символа грамматики создается свой объект.

- *LightExpression* - терминальное выражение, реализует метод Interpret() для терминальных символов грамматики. Для каждого символа грамматики создается свой объект.

- *TempExpression* - терминальное выражение, реализует метод Interpret() для терминальных символов грамматики. Для каждого символа грамматики создается свой объект.

2.2. Диаграмма последовательности

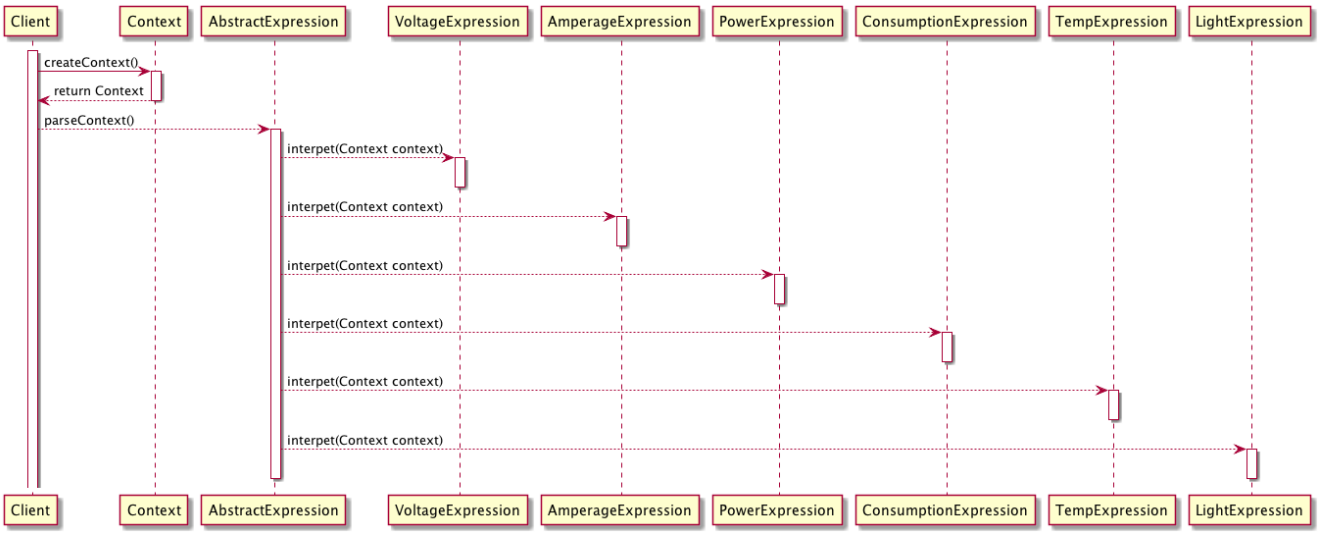

Рис. 2.2. Диаграмма последовательности паттерна Интерпретатор

Глава 3. Наблюдатель

Это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

3.1. Назначение

Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

3.2. Структура

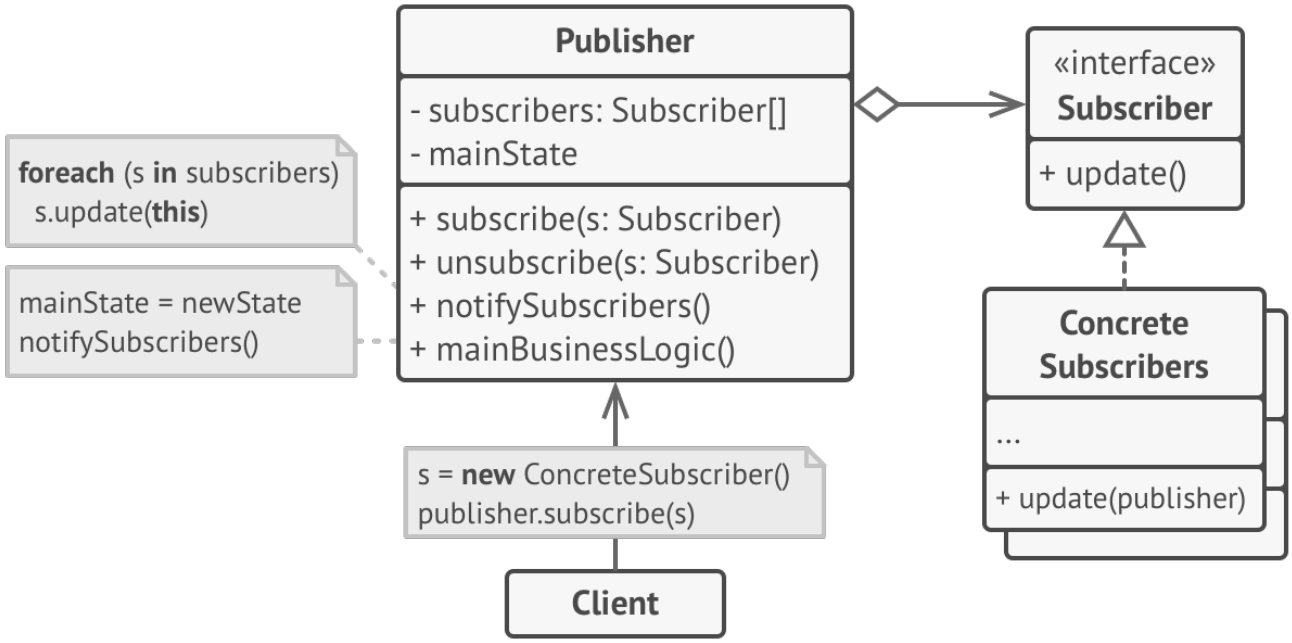

Рис. 3.1. Структура классов паттерна Наблюдатель

Участники:

- *Publisher* - Класс объекта который выполняет действия и оповещает наблюдателей.
- *Subscriber* - определяет интерфейс для обновления отправителя.

- *ConcreteSubscribers* - классы имплементирующие интерфейс Подписки и выполняют действия при обновлении.

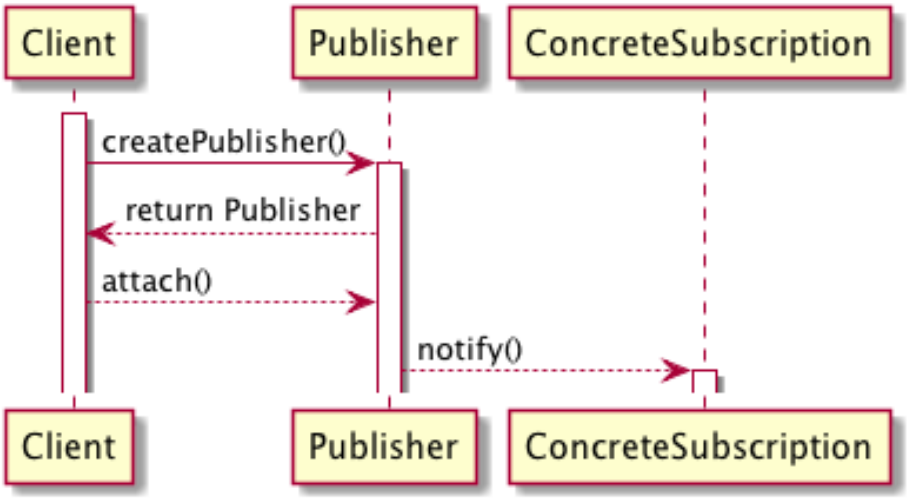

Рис. 3.2. Диаграмма последовательности паттерна Наблюдатель

3.3. Способ применения

Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.

Глава 4. Реализация паттернов

4.1. Диаграмма классов

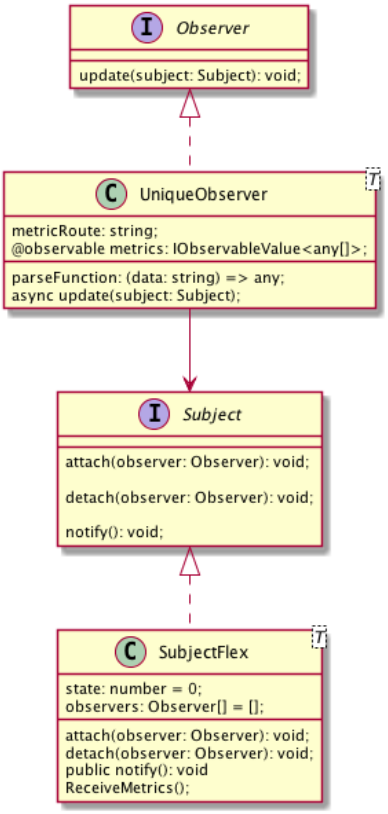

Рис. 4.1. Диаграмма классов паттерна Компоновщик

Участники:

- *Observer* - Интерфейс объекта предоставляющий функции
- *UniqueObserver* - определяет класс который будет прослушивать действия от субъекта
- *Subject* - абстрактный класс задающий логику субъекта
- *SubjectFlex* - класс задающий массив объектов и предоставляет для уведомления

4.2. Диаграмма последовательности

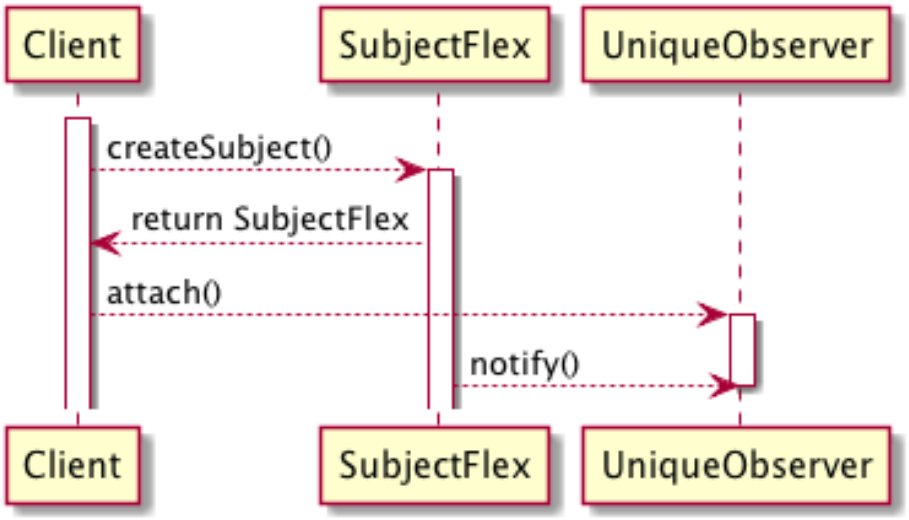

Рис. 4.2. Диаграмма последовательности паттерна Компоновщик

4.3. Код программы

```
import {EletricMetric} from './electro';
import {parseElectro, parseLight, parseTemp} from './sensor'
import {action, computed, IObservableValue, observable} from

interface Subject {
attach(observer: Observer): void;

detach(observer: Observer): void;

notify(): void;
}

class SubjectFlex<T> implements Subject {

public state: number = 0;

private observers: Observer[] = [];
```

```
public attach(observer: Observer): void {
console.log('Subject: Attached an observer.');
this.observers.push(observer);
}

public detach(observer: Observer): void {
const observerIndex = this.observers.indexOf(observer);
this.observers.splice(observerIndex, 1);
console.log('Subject: Detached an observer.');
}

public notify(): void {
console.log('Subject: Notifying observers...');
for (const observer of this.observers) {
observer.update(this);
}
}

public async ReceiveMetrics() {
this.notify();
}
}

interface Observer {
// Receive update from subject.
update(subject: Subject): void;
}

export class UniqueObserver<T> implements Observer {
metricRoute: string;
@observable metrics: IObservableValue<any[]>;
parseFunction: (data: string) => any;

constructor(route: string, parseFunction: (data: string) =>
```

```
this.metrics = observable.box([]);
this.metricRoute = route;
this.parseFunction = parseFunction;

}

@action
public async update(subject: Subject) {
fetch(`http://localhost:4000/${this.metricRoute}`, {
method: 'GET',
headers: {
'Access-Control-Allow-Origin': '*',
'Access-Control-Allow-Headers': '*'
}
}).then(item => item.json())
.then(flex => this.metrics.set([...this.metrics.get(), this.
}

@computed get Metrics() {
return this.metrics.get();
}

}


export const electronic = new SubjectFlex<any>();

export const electroObserve = new UniqueObserver<EletricMetr
export const tempObserve = new UniqueObserver<{ temp: number
export const lightObserve = new UniqueObserver<{ lux: number
//flex
electronic.attach(electroObserve);
electronic.attach(tempObserve);
electronic.attach(lightObserve);
```

```
import {EletricMetric} from './electro';

class Context<T> {
public ctx: string;
public Output: T;

constructor(ctx: string) {
this.ctx = ctx;
this.Output = <T> {};
}

}

abstract class AbstractExpression<T> {

public interpret = (context: Context<T>) => {
if (context.ctx.length == 0) {
return;
}
const idx = context.ctx.indexOf(this.MetricType());
const data = context.ctx.split(this.MetricType());
context.ctx = context.ctx.slice(idx + 1, context.ctx.length)
context.Output[this.StructType()] = parseFloat(data[0]);
console.log(context);
};

public abstract MetricType(): string;

public abstract StructType(): string;
}
```

```
class VoltageExpression extends AbstractExpression<EletricMe
MetricType() {
return 'V';
}

StructType() {
return 'voltage';
}
}

class AmperageExpression extends AbstractExpression<EletricM
MetricType(): string {
return 'A';
}

StructType(): string {
return 'amperage';
}

}

class PowerExpression extends AbstractExpression<EletricMetr
MetricType(): string {
return 'W';
}

StructType(): string {
return 'power';
}

}

class ConsumptionExpression extends AbstractExpression<Eletr
MetricType(): string {
```

```
return 'W';
}

StructType(): string {
return 'consumption';
}

}

class TempExpression extends AbstractExpression<{ temp: numb
MetricType(): string {
return 'C^';
}

StructType(): string {
return 'temp';
}

}

class LightExpression extends AbstractExpression<{ lux: numb
MetricType(): string {
return 'lux';
}

StructType(): string {
return 'light';
}

}

export const parseElectro = (data: string): any => {
const ctx = new Context<EletricMetric>(data);
const expArr = [
```

```
new VoltageExpression(),
new AmperageExpression(),
new PowerExpression(),
new ConsumptionExpression()
];
expArr.forEach(item => item.interpret(ctx));
return Dataframe(ctx.Output);
};
export const parseTemp = (data: string) => {
const ctx = new Context<{ temp: number }>(data);
const expArr = [
new TempExpression()
];
expArr.forEach(item => item.interpret(ctx));
return FrameTemp(ctx.Output);
};
export const parseLight = (data: string) => {
const ctx = new Context<{ lux: number }>(data);
console.log('parse lux');
const expArr = [
new LightExpression()
];
expArr.forEach(item => item.interpret(ctx));
console.log(ctx);
return FrameLux(ctx.Output);
};

const Dataframe = ({voltage, amperage, power, consumption}) =
({
name: generateData(), voltage, amperage, power, consumption
});

const FrameTemp = ({temp}) => ({
name: generateData(), temp
```

```
});
const FrameLux = ({light}) => ({
name: generateData(), lux: light
});

const generateData = () => {
const data = new Date();
return `${data.getMinutes()}:${data.getSeconds()}`;
};
```